

Level 1 support for CLR enums and valuetypes

© Miguel Garcia, LAMP,
École Polytechnique Fédérale de Lausanne (EPFL)
<http://lamp.epfl.ch/~magarcia>

July 20th, 2010

Abstract

At the time of this writing, Scala.NET does not emit CLR enums nor CLR valuetypes (BTW, enumerations and case classes in Scala are richer than those CLR abstractions). What Scala.NET does allow is consuming those types when defined in referenced assemblies, as well as using APIs that rely on enums and valuetypes. These notes take you on a trip from the CLR semantics of these low-level creatures, moving on to parsing of their metadata, to finally making sure that correct CIL is emitted. Some familiarity with compiler internals is assumed from the reader. The first two sections provide background on both valuetypes and the compiler components that have to know about them, sketching problems to solve. Sec. 3 covers CIL idioms for enums and valuetypes, and Sec. 4 focuses on our implementation. Unrelated to valuetypes, Sec. 5 mentions how operator overloads and conversion operators are recast from metadata to usable syntax.

Contents

1 Enumerations	2
1.1 CLR enums and CLR literal fields: What they are	3
1.2 The Scala view of .NET enumerations	4
1.3 Problem A: literal fields must be inlined	6
1.4 ‘CLR Boxing Theory’ for enums	6
1.5 Problem B: no autoboxing from <i>raw enum</i> to 0	7
1.6 More <i>peculiarities</i> of CLR enums	8
1.7 Emitting enumerations (no thanks)	8
1.8 Fine print for enum operators in other .NET languages	8
2 Valuetypes and managed pointers explained	9
2.1 How raw values come to the world	10
2.2 How managed pointers come to the world	11
2.3 Managed pointer not to valuetype but to classtype	12
2.4 Other .NET compilers and valuetypes	13
3 Things to watch out for when emitting CIL for valuetypes	14
3.1 Problem D.0: don’t try <code>ldfld</code> on <code>ref to valuetype</code>	14
3.2 Problem D.1: <code>call</code> valuetype-defined instance method	15
3.3 Problem D.2: <code>callvirt</code> valuetype-defined (virtual) methods (other than on <code>this</code>) and the <code>constrained.</code> prefix	15

3.4	Problem D.3: Accessing other non-static members on <code>this</code>	18
3.5	FYI: Assignments to dereferenced address	18
3.6	FYI: Indirect store instructions	19
3.7	FYI: when <code>GenMSIL</code> emits <code>call</code> vs. <code>callvirt</code>	19
3.8	CIL samples, enums	20
3.9	CIL samples, valuetypes	22
4	Mechanics of emitting CIL to load raw values, address values, or boxed values	23
4.1	To recap: boxing for Scala value classes	23
4.2	A trick: Use-def analysis to patch ICode instructions	24
4.3	A more elaborate trick: fine-granular types for valuetypes	24
4.3.1	Raw-valued expression found where managed pointer expected	25
4.3.2	Managed-pointer found, raw value expected (dereference)	26
4.3.3	Raw-valued expression found, boxed value expected	26
4.3.4	Managed-pointer found, boxed value expected	26
4.4	Recipe for <code>TypeParser</code>	26
4.4.1	Boxed and <code>ByRef</code>	26
4.4.2	Converting to <code>Boxed</code> and to <code>ByRef</code> when needed	27
4.4.3	When metadata refers to a <code>ByRef</code> type	27
4.4.4	Problem: implicit <code>raw2boxed</code> does not get inserted	28
4.5	Recipe for <code>GenICode</code>	28
4.5.1	Solution to <i>Raw-value found, managed-pointer expected</i>	29
4.5.2	Field selection on valuetypes (Solution D.0)	29
4.5.3	Invocations with valuetyped receiver (Solutions D.1 and D.2)	30
4.5.4	TODO <code>toString</code> vs. <code>ToString</code>	30
4.5.5	Desugar (default) constructor-invocation on valuetypes	30
4.6	Recipe for <code>GenMSIL</code>	33
5	<code>System.Decimal</code> has operator overloads and conversion operators	33
6	Sidenotes	35
6.1	How are value types implemented in the 32-bit CLR?	35
6.2	<code>GenICode</code> : valuetype class symbols are wrapped in <code>REFERENCES</code> just like those for classtypes	35
6.3	<code>GenMSIL</code> : unboxing of Booleans without <code>System.Convert</code>	35
6.4	Controlled-mutability managed pointers	36
6.5	Emitting (un)boxing in <code>GenMSIL</code>	36
6.6	“Unavoidable boxing” due to event declarations on structs	37
7	TODO	37
7.1	Passing a literal field to a byref argument should trigger an error message	37
7.2	More TODO’s	38

1 Enumerations

The idioms in `TypeParser` and `GenMSIL` for enums are representative of those for other .NET concepts, so we’ll cherry pick some problems and solutions around enum support to get us going.

1.1 CLR enums and CLR literal fields: What they are

All of the following is relevant for our discussion (quoting from §II.14.3)

1. An enum (short for enumeration) defines a set of symbols that all have the same type. A type shall be an enum if and only if it has an immediate base type of `System.Enum`. Since `System.Enum` itself has an immediate base type of `System.ValueType`, enums are value types (§13) The symbols of an enum are represented by an underlying integer type: one of `bool`, `char`, `int8`, `unsigned int8`, `int16`, `unsigned int16`, `int32`, `unsigned int32`, `int64`, `unsigned int64`, `native int`, `unsigned native int`.

[Note: Unlike Pascal, the CLI does not provide a guarantee that values of the enum type are integers corresponding to one of the symbols. In fact, the CLS (see Partition I, CLS) defines a convention for using enums to represent bit flags which can be combined to form integral value that are not named by the enum type itself. end note]

2. Enums obey additional restrictions beyond those on other value types. Enums shall contain only fields as members (they shall not even define type initializers or instance constructors); they shall not implement any interfaces; they shall have auto field layout (§10.1.2); they shall have exactly one instance field and it shall be of the underlying type of the enum; all other fields shall be static and literal (§16.1); and they shall not be initialized with the `initobj` instruction.

[Rationale: These restrictions allow a very efficient implementation of enums. end rationale]

3. The single, required, instance field stores the value of an instance of the enum. The static literal fields of an enum declare the mapping of the symbols of the enum to the underlying values. All of these fields shall have the type of the enum and shall have field `init` metadata that assigns them a value (§16.2). For binding purposes (e.g., for locating a method definition from the method reference used to call it) enums shall be distinct from their underlying type. For all other purposes, including verification and execution of code, an unboxed enum freely interconverts with its underlying type. Enums can be boxed (§13) to a corresponding boxed instance type, but this type is not the same as the boxed type of the underlying type, so boxing does not lose the original type of the enum.

Literal fields are not exclusive to enums, and require *field inlining*, §II.16.1.2:

literal specifies that this field represents a constant value; such fields shall be assigned a value. In contrast to *initonly* fields, *literal* fields do not exist at runtime. There is no memory allocated for them. *literal* fields become part of the metadata, but cannot be accessed by the code. *literal* fields are assigned a value by using the `FieldInit` syntax (§16.2).

Note: It is the responsibility of tools generating CIL to replace source code references to the literal [fields] with its actual value. Hence changing the value of a literal [field] requires recompilation of any code that references the literal [fields]. Literal [fields] are, thus, not version-resilient.

1.2 The Scala view of .NET enumerations

In `TypeParser.parseClass` the usual association between `MSILType` and `syntab.Types.Type` is tracked (i.e., the underlying type of an enumeration is not used):

```
private def parseClass(typ: MSILType) {  
  
  clrTypes.types(clazz) = typ  
  clrTypes.sym2type(typ) = clazz  
  . . .  
}
```

For an `msil.Type typ` such that `typ.IsEnum()`, the “underlying type” can be obtained with `.getUnderlyingType()`, null for non-enums.

Given that the static fields of an enum must be literal, they are subject to (mandatory) *field inlining* (Sec. 1.3),

```
.class sealed public ErrorCodes extends [mscorlib]System.Enum  
{  
  .field public unsigned int8 MyValue  
  .field public static literal valuetype ErrorCodes no_error = int8(0)  
  .field public static literal valuetype ErrorCodes format_error = int8(1)  
  .field public static literal valuetype ErrorCodes overflow_error = int8(2)  
  .field public static literal valuetype ErrorCodes nonpositive_error = int8(3)  
}
```

Although the enum fields above are initialized with integral literals, `TypeParser.parseClass` remains unimpressed by all that and ascribes those fields not a `ConstantType`, but the type of the enum:

```
val fields = typ.getFields()  
for (field <- fields if !(field.IsPrivate() || field.IsAssembly() || field.IsFamilyAndAssembly)) {  
  val flags = translateAttributes(field);  
  val name = newTermName(field.Name);  
  val fieldType =  
    if (field.IsLiteral && !field.FieldType.IsEnum) { /*- each static field of an enum IsLiteral, */  
      ConstantType(getConstant(sig2type(field.FieldType), field.getValue))  
    } else {  
      sig2type(field.FieldType) /*- yet doesn't get a ConstantType */  
    }  
  val owner = if (field.IsStatic()) statics else clazz;  
  val sym = owner.newValue(NoPosition, name).setFlag(flags).setInfo(fieldType);  
  // TODO: set private within!!! -> look at typechecker/Namers.scala  
  (if (field.IsStatic()) staticDefs else instanceDefs).enter(sym);  
  clrTypes.fields(sym) = field;  
}
```

Because of this, a value of the enum’s underlying type won’t make it for an enum-typed location (same in the other direction) (although CIL-wise it’s ok to store an integral value into an enum-typed location). This restriction is a good thing as not all values of the underlying type are allowed in general in an enum (however, one can still fabricate a non-enumerated value by AND-ing, OR-ing, etc., but this is CLR-imposed, as we’ll see shortly).

On the downside, *computing* an enum value now involves selecting one of the predefined enum’s values:

```
Z:\ea.scala:5: error: type mismatch;  
found   : scala.this.Int(5)
```

```

required: Dele.this.Color
multip.colorField = 5
      ^
Z:\ea.scala:6: error: type mismatch;
found   : Dele.this.Color
required: scala.this.Long
val age : Long = Dele.Color.Red
      ^

```

Next in `TypeParser.parseClass`, some syntax sugar is added ...

```

if (typ.IsEnum) {
  . . .
  /*- for enumerations introduce comparison and bitwise logical operations;
     the backend will recognize and replace them with comparison or
     bitwise logical operations on the primitive underlying type */
}

```

...that the backend (`GenMSIL`) promptly desugars into comparisons of integers and into bit-logical operations:

```

case CALL_METHOD(msym, style) =>
  if (msym.isClassConstructor) {
    . . .
  } else {
    . . .
    var doEmit = true
    getTypeOpt(msym.owner) match {
      case Some(typ) if (typ.IsEnum) => {
        def negBool = {
          mcode.Emit(OpCodes.Ldc_I4_0)
          mcode.Emit(OpCodes.Ceq)
        }
        doEmit = false
        val name = msym.name
        if (name eq nme.EQ) { mcode.Emit(OpCodes.Ceq) }
        else if (name eq nme.NE) { mcode.Emit(OpCodes.Ceq); negBool }
        else if (name eq nme.LT) { mcode.Emit(OpCodes.Clt) }
        else if (name eq nme.LE) { mcode.Emit(OpCodes.Cgt); negBool }
        else if (name eq nme.GT) { mcode.Emit(OpCodes.Cgt) }
        else if (name eq nme.GE) { mcode.Emit(OpCodes.Clt); negBool }
        else if (name eq nme.OR) { mcode.Emit(OpCodes.Or) }
        else if (name eq nme.AND) { mcode.Emit(OpCodes.And) }
        else if (name eq nme.XOR) { mcode.Emit(OpCodes.Xor) }
        else
          doEmit = true
      }
      case _ => ()
    }
    . . .
  /*- TODO LT, LE, GT, GE. If the underlying enum values are unsigned integers,
     then comparisons should be performed with 'cgt.un' and 'clt.un', right? */

```

The snippet above shows, for binary operators only, what gets replaced. The same technique would work for unary operators (e.g. `sizeof`) given that by the time `CALL_METHOD(msym, style) =>` as above is handled, instructions will have been emitted to place all arguments on the evaluation stack as raw (unboxed) values, the topic of the next subsection.

1.3 Problem A: literal fields must be inlined

Listing 1 on p. 41 shows how GenMSIL does literal inlining.

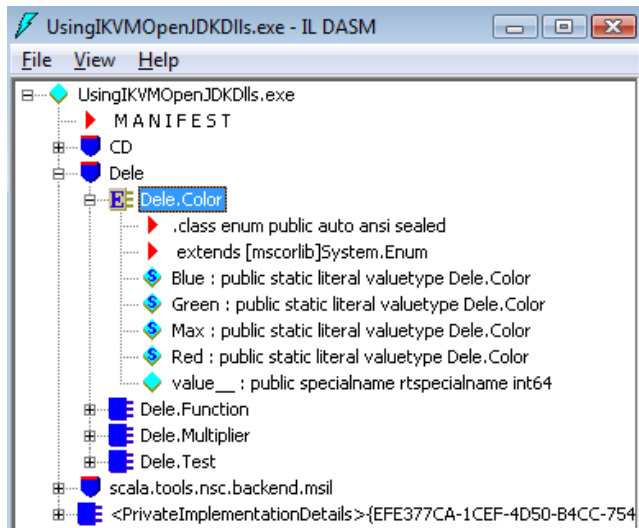
If GenMSIL generated instructions like:

```
ldsfld     valuetype [UsingIKVMOpenJDKDlls] 'Dele.Color' [UsingIKVMOpenJDKDlls] 'Dele.Color'::'Green'
```

all we would get are peverify errors like:

```
[IL]: Error: [Z:\bt.exe : bt$::main] [offset 0x0000000F]System.MissingFieldException:  
Field not found: 'Dele.Color.Green'. Field is not visible.
```

in spite of ildasm showing that Dele.Color.Green “is there”:



The CLR VM expects compilers to inline the values of (static) literal fields, in particular those of enums. Summary from §I.8.6.1.2 *Location signatures*: only static fields (of either reference, enum, or value types) can be marked as `literal`, in which case “Compilers are required to replace all references to the location with its value, and the VES therefore need not allocate space for the location.” Although they can’t be accessed, the metadata about literal fields is necessary for reflection (e.g., to list the values of an enumeration).

Note: I know the above sounds worrisome. We’re loading onto the stack an integral value for a program that was typechecked with an unboxed enum type for that value. But the CLR wants it that way and not any other.

1.4 ‘CLR Boxing Theory’ for enums

Before getting to “Problem B” (Sec. 1.5) some background is necessary on (un)boxing values of enum-underlying-type.

The spec says (§II.14.3):

For binding purposes (e.g., for locating a method definition from the method reference used to call it) enums shall be distinct from their underlying type. For all other purposes, including verification and execution of code, an unboxed enum freely interconverts with

its underlying type. Enums can be boxed (§13) to a corresponding boxed instance type, but this type is not the same as the boxed type of the underlying type, so boxing does not lose the original type of the enum.

And from §I.8.2.4:

Interfaces and inheritance are defined only on reference types. Thus, while a value type definition can specify both interfaces [to implement] and the [base] class (*System.ValueType* or *System.Enum*), these apply only to boxed values.

That explains both the C# behavior below (`colorField` is an instance field of the enum `Dele.Color`), as well as Problem B (Sec. 1.5).

```
colorField is long // false

/* Any way you look at it:
(a.1) the boxed type of colorField, i.e. Dele.Color, is not convertible to Long
(a.2) the unboxed type of colorField is also not convertible to Long,
    even with an implicit taking it to Dele.Color
*/
colorField is Dele.Color // true
colorField is System.Enum // true
colorField is System.ValueType // true
colorField is System.Object // true

/* the 2nd to last cases can be explained as:
(b.1) the boxed type of colorField, i.e. Dele.Color, is a subtype of each of
    (itself, System.Enum, System.ValueType, and System.Object)
(b.2) the unboxed type of colorField can be converted to Dele.Color, a subtype of the types listed above.
*/
```

1.5 Problem B: no autoboxing from *raw enum* to 0

With the changes to GenMSIL in Sec. 1.3, correct IL is emitted for the first line below:

```
    multip.colorField = Dele.Color.Red & Dele.Color.Green
    System.Console.WriteLine(multip.colorField)
```

However the second line does not `peverify` (“[found Long][expected ref 'System.Object'] Unexpected type on the stack.”) due to the missing `box`. For the record, `WriteLine` has type `(scala.this.Any)scala.this.Unit`, as can be seen with `-Xprint-types -Ystop:refchecks -Xprint:typer`.

```
IL_0028:  ldloc.0
IL_0029:  ldfld      valuetype [UsingIKVMOpenJDKD1ls] 'Dele.Color' [UsingIKVMOpenJDKD1ls] 'Dele.Multiplier'::'col
/*- missing: box      [UsingIKVMOpenJDKD1ls]Dele.Color */
IL_002e:  call     void [mscorlib] 'System.Console'::'WriteLine'(object)
IL_0033:  ret
```

The solution involves creating views (shown below), as covered in more detail in Sec. 4.4.

```

if (canBeTakenAddressOf) {
    // implicit conversions are owned by staticModule.moduleClass
    createViewFromTo("2Boxed", clazz.tpe, clazzBoxed.tpe, addToBoxMethodMap = true, isAddressOf = false)
    createViewFromTo("2MgdPtr", clazz.tpe, clazzMgdPtr.tpe, addToBoxMethodMap = false, isAddressOf = true)
    // a return can have type managed-pointer, thus a conversion to raw is needed
    createViewFromTo("Dereference", clazzMgdPtr.tpe, clazz.tpe, addToBoxMethodMap = false, isAddressOf = false)
    // a synthetic default constructor for raw-type allows 'new X' syntax
    if (!typ.IsEnum) {
        createDefaultConstructor(typ)
    }
}
}

```

1.6 More *peculiarities* of CLR enums

From <http://blogs.msdn.com/b/joelpob/archive/2004/07/19/187709.aspx>

The behavior of the `castclass` (or `isinst`) and `unbox` instructions over a boxed enum instance is somewhat inconsistent. Consider:

```

Color c = Color.Black;
Object o = c;
int i;
// an IL sequence using classcast will fail to assign to int local i
ldloc o
castclass int32 // this will throw InvalidCastException
stloc i
// however an unbox instruction will work just fine
ldloc o
unbox int
ldind.i4
stloc i

```

In other words there are valid and verifiable IL sequences that allow a boxed enum instance to be assigned to its underlying type or to a different compatible enum type, whereas other sequences will throw. Considering reflection always deals with boxed value, there is an asymmetry in operations that check type assignability (i.e. binding) versus operation that check instance assignability (i.e. invocation).

1.7 Emitting enumerations (no thanks)

No annotation exists to trigger the compiler to emit enumerations-metadata. There is no workaround for this, because Scala enums are richer than their .NET counterparts.

1.8 Fine print for enum operators in other .NET languages

For comparison: §14.5 *Enum values and operations* in the C# lang spec [1],

The following operators can be used on values of enum types: `==`, `!=`, `<`, `>`, `<=`, `>=` (§7.9.5), binary `+` (§7.7.4), binary `-` (§7.7.5), `^`, `&`, `|` (§7.10.2), `~` (§7.6.4), `++`, `--` (§7.5.9 and §7.6.5), and `sizeof`

(§18.5.4). Every enum type automatically derives from the class `System.Enum` (which, in turn, derives from `System.ValueType` and `object`). Thus, inherited methods and properties of this class can be used on values of an enum type.

For example:

Enumeration addition. Every enumeration type implicitly provides the following predefined operators, where E is the enum type, and U is the underlying type of E :

```
E operator +(E x, U y);  
E operator +(U x, E y);
```

The operators are evaluated exactly as $(E)((U)x + (U)y)$.

Also for comparison, F# *enum types* (which map to CLR enums) are described in §8.9 of the F# spec, http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.html#_Toc264041998

2 Valuetypes and managed pointers explained

For valuetypes and enums, a datum on the evaluation stack can be in one of three states: “value of valuetype”, “address of valuetype”, and “ref to valuetype” a.k.a. “raw value”, “managed pointer”, and “boxed value” resp. Another common term is “unboxed value” for raw value.

Some CIL instructions are choosy about which of those types they can live with. Other instructions accept more than one representation, but behave differently (e.g., invoking a mutator on a boxed representation changes the copy in the box, not the original value). This section charts those problem areas.

1. There’s no member declaration that a valuetype definition can’t contain: fields (static or instance), methods (static, instance, or virtual), properties, events, and nested types are all allowed in ILAsm.
2. Actually, there’s one member that a valuetype declaration can’t contain, a parameterless constructor.
 - We enter a synthetic default constructor for valuetypes, to enable syntax like “new V()”. The idea is for GenMSIL to translate that into a CIL `initobj` (which expects a managed pointer on the stack, Sec. 2.1)
 - See also <http://blogs.msdn.com/b/bclteam/archive/2010/04/28/constructors-and-value-types-ron-petrusha.aspx>.
3. Valuetypes are sealed, and they may implement zero or more interfaces, but this has meaning only in their boxed form.
4. Raw types are not considered subtypes of another type (except `AnyVal`) and `isinst` can be invoked only on their boxed form.

- Since valuetypes represent direct layout of data, recursive struct definitions such as (in C#) `struct S {S x; S y;}` are not permitted. A struct shall have an acyclic finite flattening graph.

Now a preposterous question: Why all the fuss about valuetypes? After all, pointers to valuetypes aren't CLS-compliant, and thus not many libraries should expose them, right? Actually, `byref` locals, params, and return types abound. That's because by default the C# compiler does not complain when emitting those and other non-CLS pearls. One has to flag a program element to be checked explicitly, for example:

```
using System;
[assembly:CLSCompliant(true)]

[CLSCompliant(true)]
public void MyMethod()
```

2.1 How raw values come to the world

One of two ways. First, using `newobj ctor`. No receiver was pushed because none is allocated yet. Afterwards, the unboxed value exists only on the stack. Taking as example a parameterful constructor `ValueType::ctor(arg1, ..., argN)`, the stack transition is:

```
..., arg1, ...argN -> ..., rawValue.
```

(As we'll see in a moment, valuetype constructors can also be invoked with `call`, in this case a managed pointer is needed below `arg1, ..., argN`).

The second way separates allocation from initialization. Raw values can be allocated as follows:

- as fields of aggregated types (classtypes or valuetypes),
- using `newarr` (for zero-based, one-dimensional arrays),
- as method arguments, or
- as local variables.

In the last case C# demands an initialization be present, e.g. `S a = default(S);` or even `S a = new S();` (even though no parameterless constructor can be defined in C# or in ILAsm for valuetypes). In both cases, `initobj typeTok` is invoked, whose stack transition is

```
. . ., destAddr -> . . .
```

Whether in C# or ILAsm, values of valuetypes can be re-initialized many times after allocation. The following is valid:

```
IL_0000: nop
IL_0001: ldloc.s a
IL_0003: initobj ReferstoUsingBlaBla.S
IL_0009: ldloc.s a
IL_000b: ldc.i4.s 20
IL_000d: call instance void ReferstoUsingBlaBla.S::setAge(int32)
IL_0012: nop
IL_0013: ldloc.s a
IL_0015: initobj ReferstoUsingBlaBla.S
```

TODO Missing in the example is a call to a custom constructor.

In short, both `initobj` and custom constructors can be called as instance methods (`call` instruction, not `newobj`) in any sequence on an (initialized or not) `valuetype` value. In both cases (`initobj`, and `call` constructor), a managed pointer stands for the receiver.

Only for ILAsm local vars, calling `initobj` can be dispensed with if `locals init` was used.

See also <http://blogs.msdn.com/b/bclteam/archive/2010/04/28/constructors-and-value-types.aspx>.

2.2 How managed pointers come to the world

Executive summary: if you want to know where `ByRefs` come to the world (which are different from `TypedByRefs`), check the following in `Type.java`:

```
/**/  
public static Type mkByRef(Type elemType) {  
    String name = elemType.FullName + "℘";  
    Type type = getType(name);  
    if (type != null) return type;  
    type = new PrimitiveType(elemType.Module,  
        TypeAttributes.NotPublic,  
        name, null, EmptyTypes, null,  
        AuxAttr.ByRef, elemType);  
    return addType(type);  
}
```

Where can byrefs be found in assembly metadata? Answer from §II.23.2 *Blobs and signatures*:

Signatures make extensive use of constant values called `ELEMENT_TYPE_xxx` – see §23.1.16. In particular, signatures include two modifiers called:

- `ELEMENT_TYPE_BYREF` – *this element is a managed pointer (see Partition I). This modifier can only occur in the definition of `LocalVarSig` (§23.2.6), `Param` (§23.2.10) or `RetType` (§23.2.11). It shall not occur within the definition of a `Field` (§23.2.4)*
- `ELEMENT_TYPE_PTR` – *this element is an unmanaged pointer (see Partition I). This modifier can occur in the definition of `LocalVarSig` (§23.2.6), `Param` (§23.2.10), `RetType` (§23.2.11) or `Field` (§23.2.4)*

Quoting from II.14.4 Pointer types

*A pointer type shall be defined by specifying a signature that includes the type of the location at which it points. A pointer can be managed (reported to the CLI garbage collector, denoted by ℘, see §14.4.2) or unmanaged (not reported, denoted by *, see §14.4.1)*

In terms of type-parsing:

```
// from PEFile.java  
public Type decodeType0() {  
    Type type = null;
```

```

int desc = readByte();
switch (desc) {
    . . .
    case ELEMENT_TYPE_PTR: // Followed by <type> token.
        if (getByte() == ELEMENT_TYPE_VOID) { // TODO
            readByte();
            type = Type.mkPtr(Type.GetType("System.Void"));
        } else type = Type.mkPtr(decodeType());
        break;
    case ELEMENT_TYPE_BYREF:
        /* although BYREF is not listed in 23.2.12. as possible alternative, this method is also called
           when parsing the signatures of a method param and a method return,
           which do allow for BYREF */
        type = Type.mkByRef(decodeType());
        break;
    case ELEMENT_TYPE_VALUETYPE: // TODO Followed by TypeDefOrRefEncoded
        assert true;
    case ELEMENT_TYPE_CLASS:
        // Followed by <type> token
        type = pemodule.getTypeDefOrRef(decodeInt());
        if (type == null) throw new RuntimeException();
        break;
    . . .
}

```

Quoting from II.14.4.2 Managed pointers

Managed pointers are specified by using $\&$ in a signature for a return value, local variable or an argument, or by using a byref type for a field or array element.

Note: the last phrase is *very* misleading (fields and array elems can't have type "managed pointer of V" and be verifiable), IMHO a better formulation is "or by passing by reference a field or array element". Fields and array elems can be type *unmanaged pointer*, rendering (the assembly?) unverifiable. No thanks.

It's possible to *take the address of a field or array element*, but not to store a managed pointer value in a field or array element, thus avoiding the possibility of dangling pointers without expensive escape analyses.

2.3 Managed pointer not to valuetype but to classtype

A method param can expect a managed pointer to valuetype, as well as to a reference type. This allows called methods to re-wire to which object a reference R refers to (at the callsite the address of R was obtained and passed as argument).

TODO What about a method returning "address of reference to classtype"? Also possible? Regarding whether that's also possible for local vars, I say: don't worry, we're not going to emit such thing. Use C++ instead.

TODO A CIL_LOAD_FIELD_ADDRESS (Sec. 4.3) is emitted for an implicit conversion to type $V\&$ with V a valuetype. In case V is a classtype, the very same CIL instruction (`ldflda FieldInfo` for a field, and so on) should be emitted. Thus no check valuetype vs. classtype is required. But checks appear necessary for another reason: obtaining the address of a field is ok as long as it corresponds to a `var` and not `val`. At the latest, this can be detected at `GenICode`. Also consider emitting a *controlled-mutability pointer* (Sec. 6.4) for the `val` case.

Note about method params in metadata:

Lidin's book, Ch. 10,

According to the common language runtime metadata model, it is not necessary to emit a Param record for each return or argument of a method. Rather, it must be done only if we want to specify the name, flags, marshaling, or default value. The IL assembler emits Param records for all arguments unconditionally and for a method return only if marshaling is specified. The name, flags, and default value are not applicable to a method return.

2.4 Other .NET compilers and valuetypes

Knowing how valuetypes behave in other languages also sheds light about differences with objects.

- <https://bugzilla.novell.com/buglist.cgi?quicksearch=valuetype>
- F# *struct types* map to CLR valuetypes, §8.8 of the F# spec, http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.html#_Toc264041997

Quoting from <http://stackoverflow.com/questions/845657/why-is-the-c-compiler-emitting-a-cal>

Summing up: Case (1) invoke virtual method: generate callvirt. Case (2) invoke instance method on nullable receiver: generate callvirt to get cheap null check – yes, this is typesafe. Case (3) invoke instance method on known non-nullable receiver: generate call to avoid null check. Your first example falls into category (2), your second example falls into category (3). (The compiler knows that new never returns null and therefore need not check again.) Eric Lippert May 11 '09 at 17:42

Quoting from the tech article *Dark corners of IronPython*¹:

- *Value Types*

Python programmer will expect all objects to behave as reference types. This rarely causes problems, but the following behavior would surprise a Python programmer:

```
>>> from System import Array
>>> from System.Drawing import Point
>>> point = Point(0, 0)
>>> array = Array[Point]((point,))
>>> array[0].X = 30
>>> array[0].X
0
```

The same thing happens in C#. Point is a struct which is a value type. When you access the first element of the array the struct is copied and the X co-ordinate is updated on the copy. When it is fetched a second time a new copy is fetched, with the original value rather than the value that was set.

- *out and ref parameters*

¹<http://www.voidspace.org.uk/ironpython/dark-corners.shtml>

Where you need to return multiple values in Python you typically do it by returning a tuple. In C# you would normally use an *out* parameter. A method that takes an *out* parameter effectively modifies the parameter in the scope calling the method. This doesn't fit with the way Python treats variables and would modify Python semantics if implemented in the same way as C#. Instead *out* parameters are simply treated as an extra return value.

```
// C#
String value;
Bool success;
success = SomeMethod(out value);
```

From IronPython: success, value = self.SomeMethod()

Similar to *out* parameters are *ref* parameters. These need to be initialised with a value, so they can't just be returned as additional values. Instead we create a reference using *clr.Reference*.

```
// C#
int value = 6;
SomeMethod(ref value);
```

```
// IronPython:
import clr
reference = clr.Reference(6)
self.SomeMethod(reference)
```

3 Things to watch out for when emitting CIL for valuetypes

Emitting CIL to access valuetype-defined static methods and static fields is a no-brainer: they are accessed the same as their classtype-defined counterparts.

3.1 Problem D.0: don't try `ldfld` on `ref` to valuetype

This is almost a non-problem, but there's a catch. Instance fields of value-types can be accessed with `ldfld`, irrespective of whether the stack top has type "valuetype" or "address of valuetype". However type "ref to valuetype" won't do.

For example, in a method with a local:

```
.locals init ([0] valuetype ReferstoUsingBlaBla.S a)
```

Each of the two instruction sequences below is valid, both behaving the same:

```
IL_0021: ldloca.s a
IL_0023: ldfld int32 ReferstoUsingBlaBla.S::age
```

```
IL_0021: ldloc.s a
IL_0023: ldfld int32 ReferstoUsingBlaBla.S::age
```

While the following fails:

```

IL_0021: ldloc.s a
           box ReferstoUsingBlaBla.S
IL_0023: ldflld int32 ReferstoUsingBlaBla.S::age
/*- [found ref 'ReferstoUsingBlaBla.S'] [expected readonly address of value 'ReferstoUsingBlaBla.S']
Unexpected type on the stack. */

```

3.2 Problem D.1: call valuetype-defined instance method

The snippets below showcase two failed attempts to invoke a valuetype-defined instance method (method `setAge(int32)`) followed by the only combination of load and call instructions that verifies.

```

// failed attempt 1
IL_0009: ldloc.s a
IL_000b: ldc.i4.s 20
IL_000d: call instance void ReferstoUsingBlaBla.S::setAge(int32)
/*- [found value 'ReferstoUsingBlaBla.S'] [expected address of value 'ReferstoUsingBlaBla.S']
Unexpected type on the stack. */

// failed attempt 2
IL_0009: ldloc.s a
           box ReferstoUsingBlaBla.S
IL_000b: ldc.i4.s 20
IL_000d: call instance void ReferstoUsingBlaBla.S::setAge(int32)
/*- [found ref 'ReferstoUsingBlaBla.S'] [expected address of value 'ReferstoUsingBlaBla.S']
Unexpected type on the stack. */

// the constrained prefix can be used only with callvirt so don't even try
// and if you try you'll get error 'Callvirt on a value type method.'

// do it this way
IL_0009: ldloca.s a
IL_000b: ldc.i4.s 20
IL_000d: call instance void ReferstoUsingBlaBla.S::setAge(int32)
/*- only way that works */

```

3.3 Problem D.2: callvirt valuetype-defined (virtual) methods (other than on this) and the constrained. prefix

Virtual methods (implementations of interface methods must be virtual) expect a managed pointer for `this`, but can also be given a boxed value. And, oh, they can also be invoked with the `constrained.` prefix.

In case one of these overrides behaves as a mutator, and the invocation has a boxed value as receiver, it will mutate the copy in the boxed representation, not the original data location. This is expected, across .NET languages:

```

/*- C# code */
public struct S {
    public int age;
    public override String ToString() {
        age++;
        return "age == " + age + ", after mutating.";
    }
}

```

```

class Program {
    static void Main(string[] args) {
        S a = new S();
        a.setAge(20);
        object boxed = a;
        System.Console.WriteLine(boxed.ToString()); // prints age == 21
        System.Console.WriteLine(a.age); // prints age == 20
    }
}

```

Note: none of the methods in `System.Object`, `System.ValueType`, or `System.Enum` mutate the receiver.

From a GenMSIL perspective, the following example highlights two things: (a) no superfluous boxing should be introduced; and (b) a managed pointer should be pushed as receiver for `ToString()` (which as we know is a virtual method but please don't box just to use `callvirt`).

```

/*- C# code */
static void Main(string[] args)
{
    S a = new S();
    a.setAge(20);
    System.Console.WriteLine(a.ToString()); // prints 21
    System.Console.WriteLine(a.age); // prints 21
}

```

The C# compiler emits the following CIL for the statement `System.Console.WriteLine(a.ToString());`:

```

/*- WAY 1 of 2 of invoking a valuetype-defined virtual method without boxing */
IL_0013: ldloca.s a
IL_0015: constrained. ReferstoUsingBlaBla.S
IL_001b: callvirt instance string [mscorlib]System.Object::ToString()
IL_0020: call void [mscorlib]System.Console::WriteLine(string)

```

Although the overriding method could also be called like this:

```

/*- WAY 2 of 2 of invoking a valuetype-defined virtual method without boxing */
IL_0013: ldloca.s a
IL_001b: call instance string ReferstoUsingBlaBla.S::ToString() /*- not System.Object */

```

or even as follows, but only at the cost of mutating the boxed copy and not the original:

```

IL_0013: ldloc.s a
           box ReferstoUsingBlaBla.S
IL_001b: callvirt instance string System.Object::ToString()
/*- none of the methods in System.Object, System.ValueType, or System.Enum mutate the receiver,
but this example shows that the boxed copy gets mutated */

```

It's interesting to note that irrespective of how control flow reaches the overriding method ("WAY 1 of 2", "WAY 2 of 2", or using boxing as above) the type of `arg0` in the body is "address of valuetype".

In other words: in the body of any valuetype-defined non-static method (yes, instance or virtual) the type of `arg0` is "address of valuetype".

It's a bit surprising in the third case, where a boxed value (of type `ref to valuetype`) was pushed at the callsite. This means that, were we to invoke

another method from that method, the following would work:

```
ldarg.0
ldc.i4.s 2
call instance void ReferstoUsingBlaBla.S::setAge(int32)
```

but not this

```
ldarg.0
callvirt instance int32 [mscorlib]System.Object::GetHashCode()
pop
/*-
[IL]: Error: [found address of value 'ReferstoUsingBlaBla.S'] [expected ref 'System.Object'] Unexpected type on
[IL]: Error: Call to base type of valuetype.
*/
```

which is otherwise perfectly fine (i.e., perfectly fine in a *class-defined* virtual method). The lesson here is not that a virtual method can't be called virtually, but that the method signature given must be that for the overriding method.

Note: If you're into AOP for .NET and can spot when a joinpoint will capture one but not the other of ("WAY 1 of 2", "WAY 2 of 2") please let me know.

Some do's and don't's about constrained.

```
// failed attempt 1
IL_0013: ldloc.s a
        box ReferstoUsingBlaBla.S
IL_0015: constrained. ReferstoUsingBlaBla.S
IL_001b: callvirt instance string System.Object::ToString()
/*- Error: The 'this' argument to a constrained call must have ByRef type */

// failed attempt 2
IL_0013: ldloc.s a
IL_0015: constrained. ReferstoUsingBlaBla.S
IL_001b: callvirt instance string ReferstoUsingBlaBla.S::ToString()
/*- Callvirt on a value type method. */

// this way works
IL_0013: ldloc.s a
IL_0015: constrained. ReferstoUsingBlaBla.S
IL_001b: callvirt instance string System.Object::ToString()
```

Note: Before .NET 2.0 (when the `constrained.` didn't exist) more methods of valuetypes were invoked using `call` rather than `callvirt` as nowadays. Details at <http://doogalbellend.blogspot.com/2007/04/method-calls-on-value-types-and-boxing.html>

The `constrained.` prefix also comes into play (principally actually) in connection with invocations on a type parameter that is not known for sure to be instantiated with either an object type or a valuetype. An example at <http://bartdesmet.net/blogs/bart/archive/2007/02/20/fun-with-generics-about-the-new-constraint-and.aspx>

3.4 Problem D.3: Accessing other non-static members on this

Self-quoting from Sec. 3.3:

In other words: in the body of any valuetype-defined non-static method (yes, instance or virtual) the type of `arg0` is “address of valuetype”.

TODO How to set the self-type in valuetypes to “address of valuetype”?

So far, bytecode to load `this` is generated as follows:

```
case THIS(clasz) =>
    mcode.Emit(OpCodes.Ldarg_0)
. . .
case STORE_THIS(_) =>
    // this only works for impl classes because the self parameter comes first
    // in the method signature. If that changes, this code has to be revisited.
    mcode.Emit(OpCodes.Starg_S, 0)
```

3.5 FYI: Assignments to dereferenced address

This is off-topic because the assemblies emitted by Scala.NET aim at CLS-compliance (e.g., no byref-params in exported interfaces) and thus the need will not arise to emit CIL as discussed in this subsection. It’s instructive however to see how non-CLS-compliant methods use values received byref, methods that Scala.NET can invoke.

In C#, a method receiving a byref parameter can appear as LHS in an assignment. The RHS is usually a raw value, but can also be another managed pointer.

In CIL, indirect stores always expect a raw value as source, thus a C# RHS with address-type is desugared first into an indirect load of the RHS expression. The `stobj` instruction can duplicate `stind` functionality, but its specialty are generic parameters.

The resulting CIL idioms appear below.

```
public static void demo(ref Color rc, ref DBBool rb1, ref DBBool rb2)
{
    rc = Color.Red;
    rb1 = new DBBool();
    rb2 = DBBool.Null;
    rb1 = rb2;
}
```

```
.method public hidebysig static void demo(valuetype ReferstoUsingBlaBla.Color& rc,
                                           valuetype ReferstoUsingBlaBla.DBBool& rb1,
                                           valuetype ReferstoUsingBlaBla.DBBool& rb2) cil managed
{
    // Code size      35 (0x23)
    .maxstack 8
    IL_0000: nop

    IL_0001: ldarg.0
    IL_0002: ldc.i4.0
    IL_0003: stind.i4 /*- rc = Color.Red; */
    IL_0004: ldarg.1
```

```

IL_0005: initobj    ReferstoUsingBlaBla.DBBool /*- rb1 = new DBBool(); */
IL_000b: ldarg.2
IL_000c: ldsfld    valuetype ReferstoUsingBlaBla.DBBool ReferstoUsingBlaBla.DBBool::Null
IL_0011: stobj    ReferstoUsingBlaBla.DBBool /*- rb2 = DBBool.Null; */

IL_0016: ldarg.1
IL_0017: ldarg.2
IL_0018: ldobj    ReferstoUsingBlaBla.DBBool
IL_001d: stobj    ReferstoUsingBlaBla.DBBool /*- rb1 = rb2; */

IL_0022: ret
} // end of method DBInt::demo

```

3.6 FYI: Indirect store instructions

stind.i1 , stind.i2 , stind.i4 , stind.i8 , stind.r4 , stind.r8 , stind.i , stind.ref are covered in Figure 2 on p. 39. And the stobj story appears in Figure 3 on p. 40.

3.7 FYI: when GenMSIL emits call vs. callvirt

How does GenMSIL currently choose emitting call or callvirt? We catalog the different scenarios. Not that a call being emitted for a CLR virtual method would fail on a valuetype (method lookup picks the most overridden version anyway, i.e., that in the valuetype). This catalog just reveals what call style happens behind the scenes.

- In case an entry point was found, BytecodeGenerator.writeAssembly() synthesizes a global method to become the new entry point. This method callvirt the original entry point, which was defined as non-static on a module.
- for string concatenation (as part of genPrimitive())
- as part of dumpMirrorClass()
- as part of createDelegateCaller()

TODO There are also several places in GenMSIL where mcode.Emit(OpCodes.Callvirt, ...) shows up, it would be great just to be safe to take a look at them.

At the end of case CALL_METHOD => in GenMSIL (after special cases have been handled), call or callvirt are emitted for all other method invocations depending on:

```

if (doEmit) {
    val methodInfo: MethodInfo = getMethod(msym)
    (style: @unchecked) match {
        case SuperCall(_) =>
            mcode.Emit(OpCodes.Call, methodInfo)
        case Dynamic =>
            mcode.Emit(if (dynToStatMapped(msym)) OpCodes.Call else OpCodes.Callvirt,
                methodInfo)
        case Static(_) =>

```

```

        mcode.Emit(OpCodes.Call, methodInfo)
    }
}

```

Summing up, in this last scenario:

1. `call` is always emitted for static and super invocations. Some static invocations have a `this` instance (see below) but i-regardless `call` is emitted;
2. `call` is also emitted for those Java instance methods that `BytecodeGenerator.initMappings()` “maps” to .NET static counterparts; and
3. `callvirt` for the rest (i.e., the majority).

```

/** Virtual calls */
case object Dynamic extends InvokeStyle

/** InvokeDynamic a la JSR 292 (experimental). */
case object InvokeDynamic extends InvokeStyle

/**
 * Special invoke. Static(true) is used for calls to private members. */
case class Static(onInstance: Boolean) extends InvokeStyle

/** Call through super[mix]. */
case class SuperCall(mix: Name) extends InvokeStyle

```

Also relevant:

```

/** This class represents a CALL_METHOD instruction
 * STYLE: dynamic / static(onInstance == true)
 * Stack: ...:ref:arg1:arg2:...:argn
 * ->: ...:result
 *
 * STYLE: static(onInstance == false)
 * Stack: ...:arg1:arg2:...:argn
 * ->: ...:result
 *
 */
case class CALL_METHOD(method: Symbol, style: InvokeStyle) extends Instruction {

```

There’s a third invocation style, `InvokeDynamic` (a la JSR 292) that shouldn’t show up in `forMSIL`.

TODO Question about the then-branch below: who pops the superflous `this` instance that will go unused?

```

mcode.Emit(if (dynToStatMapped(msym)) OpCodes.Call else OpCodes.Callvirt,
           methodInfo)

```

3.8 CIL samples, enums

```

public enum Color
{
    Red, Green, Blue
}
public class ColorDemo

```

```

{
    public static void rcvColor(ref Color rc) { rc = Color.Blue; }
}
public class MoreColorDemo
{
    public readonly Color finalColor = Color.Red;
    public static void main()
    {
        Color c = Color.Red;
        /*readonly*/ Color d = c & Color.Green; // C# disallows readonly local variables
        System.Console.WriteLine(d);
        Color w = new Color();
        ColorDemo.rcvColor(ref c); // but not ref Color.Red
        ColorDemo.rcvColor(ref d); // but not finalColor
    }
}

```

ColorDemo.rcvColor:

```

.method public hidebysig static void rcvColor(valuetype Color& rc) cil managed
{
    // Code size      5 (0x5)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldarg.0
    IL_0002: ldc.i4.2
    IL_0003: stind.i4
    IL_0004: ret
} // end of method ColorDemo::rcvColor

```

And the main driver program:

```

.method public hidebysig static void main() cil managed
{
    // Code size      38 (0x26)
    .maxstack 2
    .locals init ([0] valuetype Color c,
        [1] valuetype Color d,
        [2] valuetype Color w)
    IL_0000: nop
    IL_0001: ldc.i4.0
    IL_0002: stloc.0 /*- Color c = Color.Red; */
    IL_0003: ldloc.0
    IL_0004: ldc.i4.1
    IL_0005: and
    IL_0006: stloc.1 /*- Color d = c & Color.Green; */
    IL_0007: ldloc.1
    IL_0008: box      Color
    IL_000d: call     void [mscorlib]System.Console::WriteLine(object) /*- System.Console.WriteLine(d); */
    IL_0012: nop
    IL_0013: ldc.i4.0
    IL_0014: stloc.2 /*- Color w = new Color(); */
    IL_0015: ldloca.s  c
    IL_0017: call     void ColorDemo::rcvColor(valuetype Color&) /*- ColorDemo.rcvColor(ref c); */
    IL_001c: nop
    IL_001d: ldloca.s  d
    IL_001f: call     void ColorDemo::rcvColor(valuetype Color&) /*- ColorDemo.rcvColor(ref d); */
    IL_0024: nop
    IL_0025: ret
} // end of method MoreColorDemo::main

```

3.9 CIL samples, valuetypes

```
public struct X
{
    public Int32 x;
    public void v() { }
    public X(Int32 x) { this.x = x; }
}
```

And the main driver program, XDemo:

```
public class XDemo
{
    public static readonly X fvA = new X();
    public static readonly X fvB = new X(42);

    public static void rcvX(ref X rx) { rx = fvB; }

    public static void main()
    {
        X c = fvA;
        System.Console.WriteLine(c);
        rcvX(ref c); // but not ref fvA
    }
}
```

First, rcvX

```
.method public hidebysig static void rcvX(valuetype X& rx) cil managed
{
    // Code size      13 (0xd)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldarg.0
    IL_0002: ldsfld    valuetype X XDemo::fvB
    IL_0007: stobj    X
    IL_000c: ret
} // end of method XDemo::rcvX
```

CIL for the main test:

```
.method public hidebysig static void main() cil managed
{
    // Code size      28 (0x1c)
    .maxstack 1
    .locals init ([0] valuetype X c)
    IL_0000: nop
    IL_0001: ldsfld    valuetype X XDemo::fvA
    IL_0006: stloc.0
    IL_0007: ldloc.0
    IL_0008: box      X
    IL_000d: call    void [mscorlib]System.Console::WriteLine(object)
    IL_0012: nop
    IL_0013: ldloca.s c
    IL_0015: call    void XDemo::rcvX(valuetype X&)
    IL_001a: nop
    IL_001b: ret
} // end of method XDemo::main
```

4 Mechanics of emitting CIL to load raw values, address values, or boxed values

Previous sections spell out the issues around raw values, managed pointers, and boxed values. This section covers the changes in `TypeParser`, `GenICode`, and `GenMSIL` necessary to address those issues. Before getting there, we recap how boxing is used for primitives. The current solution is described in Sec. 4.3, while Sec. 4.2 covers another approach which was not followed.

4.1 To recap: boxing for Scala value classes

```
// Definitions.scala
```

```
def ScalaValueClasses = List(  
  UnitClass, ByteClass, ShortClass, IntClass, LongClass,  
  CharClass, FloatClass, DoubleClass, BooleanClass  
)
```

`GenMSIL` emits boxing in only two cases:

- upon visiting an `ICode` stmt `BOX(sourceTypeKind)`
case `BOX(boxType) => emitBox(mcode, boxType)`
- as part of `createDelegateCaller`

Similarly for unboxing (i.e., upon visiting `UNBOX(toTypeKind)`, and as part of `createDelegateCaller`).

`GenICode` in turn inserts `BOX` stmts only when a method marked `isBox` receives a single argument (the boxable-typed expression `to box`):

```
case Apply(fun @ _, List(expr)) if (definitions.isBox(fun.symbol)) =>
```

where `definitions.isBox(msym)` just looks up the method symbol among `boxMethod`'s values. (Similarly, `GenICode` emits unboxing for expressions of shape as above where `definitions.isUnbox(fun.symbol)`).

In both cases, the `BOX` and the `UNBOX` instructions carry as argument the pre-instruction type on the stack (the source or *raw* type for `INBOX`, and the boxed version for `UNBOX`). These also correspond to the `typeTok` expected by CIL `box` and `unbox` instructions.

However, `ICode`-level boxing is not the only means to emit CIL (un)boxing. Additionally, one may enter (synthetic) implicit (static) conversion methods, and have `GenMSIL` special-case their handling by `case CALL_METHOD` in a type-safe manner (the emitted CIL (un)boxing instruction expects the same type-stack and leaves the same type-stack as the original `case CALL_METHOD` instruction).

- The technique above works when the expected type is a boxed value, which can be obtained from either a raw value or a managed pointer already on the evaluation stack (by emitting CIL `box V` in the former case and `ldobj V; box V` in the second).
- It also works to obtain a raw value from a managed pointer on the stack, i.e. `ldobj V`

However it does not work when the conversion-result should be a managed pointer. For that, we need another technique. Two candidates are described next (Sec. 4.2 and Sec. 4.3).

4.2 A trick: Use-def analysis to patch ICode instructions

This trick sounds almost too good to be true. So caution is advised.

Rather than making `TypeParser` distinguish between the raw-type, boxed-type, and reference-to-raw-type representations that a valuetype `V` gives origin to, let it blissfully ignore those differences, regarding `V` as a classtype, just another subtype from `System.Object`.

To do away with ensuing `peverify` errors like ‘ref expected, value found’ adapt an existing use-def analysis (*Using reaching-defs and type-flow analyses to obtain three-address code in the Scala compiler*²) to detect for example a valuetype-pushing instruction `Instr` whose `next` should be `BOX(V)`.

`Instr` can be one of:

- `LOAD_FIELD`, which `GenMSIL` will later inline by emitting `ldc`. Box the primitive on the stack.
- `CALL_PRIMITIVE`, this case results from an enum operator. Box the result on the stack.
- `CALL_METHOD`, i.e. a method returns a value of the underlying type. Box it with `BOX(V)`.
- `DUP`. See remarks below.

TODO: What if <code>Instr</code> is <code>DUP</code> ? A use-def pointing to <code>DUP</code> as def means only the topmost value in the stack should be boxed. If the load instruction (immediately before) the <code>DUP</code> is pointed as def, then the next-to-topmost is to be boxed. In the latter case, we can’t box and then <code>DUP</code> if the duffed value is going to be used unboxed
--

TODO Not clear whether this trick will reliably achieve all needed conversions, while not inserting superfluous boxing (which leads to mutations affecting temporary values and thus getting lost)
--

4.3 A more elaborate trick: fine-granular types for value-types

At the assembly-metadata level there is just one declaration for a valuetype `v` (conceptually describing the boxed formulation) that is referred from both constructs expecting raw values (e.g., a `v` field) and those expecting managed pointers (e.g., a `v&` param).

Given a metadata valuetype `V` we enter type symbols for each of raw-values, managed-pointers, and boxed-values of `V`.

- in terms of Scala programs, only the “raw-values type” can be mentioned in user programs (it has the same name as `v`). The two others have un-speakable names.

²<http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q2/threaddress.pdf>

- a type symbol for managed pointers is needed because constructs in externally-defined assemblies mention it (in params and returns) (and we want to consume those assemblies from Scala).
- boxed values can live only on the evaluation stack, and therefore no meta-data construct has that type (nor can .NET programs declare variables of that type, no, not even ILAsm). If that were the full story, Scala.NET could do without a type symbol for them. However, valuetypes can implement interfaces, and virtual methods in general (whether overriding an interface member or locally declared) can be invoked only on a boxed receiver. Therefore, an implicit conversion from raw value into `System.Object` is too coarse.

A raw value on the stack tells us nothing about the location it came from. In case a raw value of `V` appears where `V&` is expected, the usual process by which GenICode emits `LOAD_FIELD`, `LOAD_LOCAL`, and `LOAD_ARRAY_ITEM` has to be customized as follows.

4.3.1 Raw-valued expression found where managed pointer expected

Taking `LOAD_FIELD` as an example, let's assume a field `sf` has been declared of valuetype `V` and a Scala snippet where `sf` has to be passed "by reference" to a method expecting `V&`. At GenICode, the input contains a subexpression along the lines of "`convertV2AddressOfV(sf)`" and the desired CIL is "`ldflda sf-FieldInfo`", i.e. we want to elide the invocation to the marker method, generating an ICode instruction, `CIL_LOAD_FIELD_ADDRESS` for the whole tree "`convertV2AddressOfV(sf)`" (the translation of the receiver of that `Select` is another matter).

1. Currently, `genLoad` in GenICode already showcases how an input tree with shape "`callsite(arg)`" is detected (for the purpose of replacing the "`callsite`" with a `BOX` instruction), Listing 2. The differences with our goal are described in the next item.
2. In our case, (a) we emit no instruction for the call, after recognizing it by looking up in a dedicated set, and (b) the tree for the argument to the call is explored to detect whether it's any of:
 - (a) `Select(qualifier, selector)`, which may be translated with `genLoadModule` (in case `tree.symbol.isModule`), or as a `LOAD_FIELD` instruction. This last case we want to handle our way, emitting instead `LOAD_FIELD_ADDRESS`. TODO: what about getters.
 - (b) similarly for those places where `LOAD_LOCAL` is emitted
 - (c) and for `genArrayOp`, the only place where the string "`.emit(LOAD_ARRAY_ITEM()`" shows up in GenICode
 - (d) trees not matching any of the shapes above denote a computation of a raw value of `V` as intermediate result, we can still take its address (with CIL `box V`; `unbox V`). The user should be aware that mutations on the intermediate value will get lost. Of the two previous CIL instructions, the first one results from emitting an ICode `BOX(V)` instruction. For the second instruction, ICode `UNBOX` is not what we

need (GenMSIL emits `unbox V`; `ldobj V` for it). A new ICode instruction (termed `CIL.UNBOX(V)`) is needed to be translated by GenMSIL into `unbox V`.

4.3.2 Managed-pointer found, raw value expected (dereference)

This case results from invoking an externally-defined method returning “by reference” a managed pointer `mp`. Instance methods can be invoked on `mp`. Additionally, an implicit can be defined to convert `mp` to raw value (say, for assignment). Such implicit is translated by GenMSIL just to `ldobj V`.

4.3.3 Raw-valued expression found, boxed value expected

A boxed value is needed in a few situations:

- a `System.Object` is needed. For example: `System.Console.WriteLine(sf)`. First, an implicit conversion expands to `System.Console.WriteLine(convertV2BoxedV(sf))`, and the conversion’s invocation is later desugared by GenMSIL into `CIL box V` (Sec. 6.5).
- a boxed value (and only a boxed value) is needed. This may be due to a number of factors. In all cases, the technique above also works here.
 - a virtual method is to be invoked on the the boxed value.
 - the raw value is assigned to a location typed with one of `V`’s supported interfaces

4.3.4 Managed-pointer found, boxed value expected

Do we want to support this? It’s only needed if a by-ref returning method is invoked, say, as argument to `System.Console.WriteLine()`. A workaround consists in assigning first the method result to a raw-valued local (relying on the dereferencing conversion). The code is quite readable this way, thus why bother to add yet another implicit? If for some reason we’re already emitting `ldobj V`; `box V` then maybe it’s ok to special-case this conversion, otherwise I suggest the workaround.

4.4 Recipe for TypeParser

4.4.1 Boxed and ByRef

Given a valuetype `V` in metadata, we enter type symbols for each of raw-values, managed-pointers (“`VMgdPtr`”), and boxed-values (“`VBoxed`”). This happens here:

```
val canBeTakenAddressOf = (typ.IsValueType || typ.IsEnum) && (typ.FullName != "System.Enum")
if (canBeTakenAddressOf) {
    clazzBoxed = clazz.owner.newClass(clazz.name + "Boxed")
    clazzMgdPtr = clazz.owner.newClass(clazz.name + "MgdPtr")
    /*- adding typMgdPtr to clrTypes.sym2type should happen early (before metadata for supertypes is parsed,
       before metadata for members are parsed) so that clazzMgdPtr can be found by sig2type. */
    val typMgdPtr = MSILType.mkByRef(typ)
    clrTypes.types(clazzMgdPtr) = typMgdPtr
    clrTypes.sym2type(typMgdPtr) = clazzMgdPtr
}
```

```

val instanceDefsMgdPtr = new Scope
val classInfoMgdPtr = ClassInfoType(definitions.anyvalparam, instanceDefsMgdPtr, clazzMgdPtr)
clazzMgdPtr.setFlag(flags)
clazzMgdPtr.setInfo(classInfoMgdPtr)
}

```

Only the “raw values type” can be ascribed in user programs (it has the same name as `V`) (the two others should have unspeakable names).

4.4.2 Converting to Boxed and to ByRef when needed

Shortly after, three views are entered (from raw to managed pointer and to boxed; also from managed pointer to raw value). Their sole purpose is hinting `GenICode` and `GenMSIL` to insert the appropriate CIL instruction sequences to realize boxing/address-taking/dereferencing, as we’ll see later.

4.4.3 When metadata refers to a ByRef type

During metadata parsing, we may encounter a usage of the byref version of a valuetype `V` which hasn’t been parsed yet. Given that we instantiate a class symbol for `V&` in `TypeParser.parseClass()` only, `sig2type` has to know how to have `V` *completed* when the type symbol for `V&` is requested:

```

private def sig2type(tMSIL: MSILType): Type = {
  var res = getCLRTypeIfPrimitiveOrNullOtherwise(tMSIL)
  if (res != null) res
  else if (tMSIL.isInstanceOf[ConstructedType]) {
    . . .
  } else if (tMSIL.isInstanceOf[TMVarUsage]) {
    . . .
  } else if (tMSIL.IsArray()) {
    . . .
  } else {
    res = clrTypes.sym2type.get(tMSIL) match {
      case Some(sym) => sym.tpe
      case None => if (tMSIL.IsByRef && tMSIL.GetElementType.IsValueType) {
        val addressed = sig2type(tMSIL.GetElementType)
        val clasym = addressed.typeSymbolDirect // TODO should be .typeSymbol?
        clasym.info.load(clasym)
        val secondAttempt = clrTypes.sym2type.get(tMSIL)
        secondAttempt match { case Some(sym) => sym.tpe
                              case None => null
                            }
      } else getClassType(tMSIL)
    }
  }
  if (res == null)
    . . .
  else res
}
}

```

TODO So far `sig2type` can answer requests for `V&` with `V` a valuetype but not those where `V` is a type parameter, e.g. `0!&`. An example of this is the `System.Enum.TryParse<TEnum>` method:

```

public static bool TryParse<TEnum>(

```

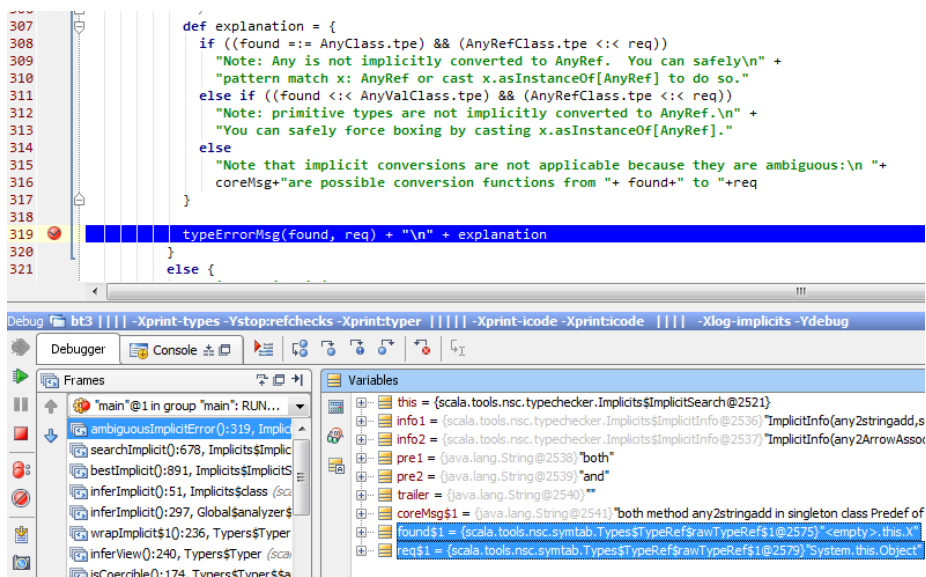


Figure 1: raw2boxed does not get inserted

```

    string value,
    out TEnum result
)
where TEnum : struct, new()

```

4.4.4 Problem: implicit raw2boxed does not get inserted

For now I'm adding it manually. Example: `System.Console.WriteLine(X.view2Boxed(v1))`.

Two other views are found instead:

both method any2stringadd in singleton class Predef of type (<param> x: scala.this.Any)runtime.this.StringAdd and method any2ArrowAssoc in singleton class Predef of type [<deferred> <param> A](<param> x: A)scala.this.Predef

Because of the test:

```
if ((found <:< AnyValClass.tpe) && (AnyRefClass.tpe <:< req))
```

the following error message is shown (although our X is not a primitive). Debug view details at Figure 1.

Note: primitive types are not implicitly converted to AnyRef. You can safely force boxing by casting `x.asInstanceOf[AnyRef]`.

Also helpful:

```
val XlogImplicits = BooleanSetting ("-Xlog-implicits",
    "Show more info on why some implicits are not applicable")
```

4.5 Recipe for GenICode

GenICode knows about several CLR-specific instructions, all dealing with man-

aged pointers in one way or another:

```
// from Opcodes.scala
case class CIL_LOAD_LOCAL_ADDRESS(local: Local) extends Instruction { . . . }

case class CIL_LOAD_FIELD_ADDRESS(field: Symbol, isStatic: Boolean) extends Instruction { . . . }

case class CIL_LOAD_ARRAY_ITEM_ADDRESS(kind: TypeKind) extends Instruction { . . . }

case class CIL_UNBOX(boxType: TypeKind) extends Instruction { . . . }

case class CIL_INITOBJ(valueType: TypeKind) extends Instruction { . . . }

case class CIL_NEWOBJ(method: Symbol) extends Instruction { . . . }
```

4.5.1 Solution to *Raw-value found, managed-pointer expected*

The first three instructions above are emitted in `genLoadAddressOf`, a new utility method in `GenICode` that handles the cases mentioned in Sec. 4.3.1 to effect conversions from raw value to managed pointer are inserted. Those callsites are desugared into what `genLoadAddressOf()` emits:

```
// from GenICode.scala
case Apply(fun @ _, List(expr)) if (loaders.clrTypes.isAddressOf(fun.symbol)) =>
  if (settings.debug.value)
    log("ADDRESSOF : " + fun.symbol.fullName);
  val ctx1 = genLoadAddressOf(expr, ctx, toTypeKind(expr.tpe))
  generatedType = toTypeKind(fun.symbol.tpe.resultType)
  ctx1
```

4.5.2 Field selection on valuetypes (Solution D.0)

As we saw in Sec. 3.1, `ldfld` will verify as long as the stack contains a raw value or a managed pointer for the expected type. By default, a raw-value will be loaded by:

```
// from GenICode.scala
case Select(qualifier, selector) =>
  val sym = tree.symbol
  generatedType = toTypeKind(sym.info)

  if (sym.isModule) {
    if (settings.debug.value)
      log("LOAD_MODULE from Select(qualifier, selector): " + sym)
    assert(!tree.symbol.isPackageClass, "Cannot use package as value: " + tree)
    genLoadModule(ctx, sym, tree.pos)
    ctx
  } else if (sym.isStaticMember) {
    ctx.bb.emit(LOAD_FIELD(sym, true), tree.pos)
    ctx
  } else {
    val ctx1 = genLoadQualifier(tree, ctx) /*- for sym.owner a valuetype, loads raw value, not address */
    ctx1.bb.emit(LOAD_FIELD(sym, false), tree.pos)
    ctx1
  }
}
```

Irrespective of the field's type, the value that `LOAD_FIELD` in turn loads onto the stack is the same, whether `genLoadQualifier` loaded a raw value (as done now) or an address. Therefore, we don't mess with the current functionality as it's ok as is (internally, the runtime copies the address for large valuetypes, Sec. 6.1, so no performance is lost).

4.5.3 Invocations with valuetype receiver (Solutions D.1 and D.2)

Summing up sections 3.2 and 3.3, we adopt the idioms below when invoking valuetype-defined methods:

```
// instance method
IL_0009: ldloca.s a
IL_000b: ldc.i4.s 20
IL_000d: call instance void ReferstoUsingBlaBla.S::setAge(int32)
/*- only way that works */
```

```
/*- WAY 1 of 2 of invoking a valuetype-defined virtual method without boxing */
IL_0013: ldloca.s a
IL_0015: constrained. ReferstoUsingBlaBla.S
IL_001b: callvirt instance string [mscorlib]System.Object::ToString()
```

TODO

4.5.4 TODO toString vs. ToString

There's a quirk in that:

```
System.Console.WriteLine(X.view2Boxed(v1).toString)
```

typechecks all right, but if one writes `X.view2Boxed(v1).ToString` then one sees:

```
error: value ToString is not a member of <empty>.this.XBoxed

qual = X.view2Boxed(v1):<empty>.this.XBoxed
Symbol=<none>
symbol-info = <notype>
scope-id = 22516593
members = List(method hashCode, method equals, method toString, constructor ValueType, method notifyAll, method
name = ToString
found = <none>
owner = singleton class bt2
```

4.5.5 Desugar (default) constructor-invocation on valuetypes

Sec. 2.1 covers the CIL idioms to obtain a raw value on the stack. Default constructors are added by `TypeParser` to non-enum valuetypes, just after adding the implicit conversions (Sec. 1.5 on p. 7). After invocation, the new values usually appear as RHS in `Assign(lhs, rhs)` statements or in `ValDef` statements, which are translated by `GenICode` as follows (FYI):

```
private def genStat(tree: Tree, ctx: Context): Context = tree match {
  case Assign(lhs @ Select(_, _), rhs) =>
    val isStatic = lhs.symbol.isStaticMember
```

```

var ctx1 = if (isStatic) ctx else genLoadQualifier(lhs, ctx)

ctx1 = genLoad(rhs, ctx1, toTypeKind(lhs.symbol.info))
ctx1.bb.emit(STORE_FIELD(lhs.symbol, isStatic), tree.pos)
ctx1

case Assign(lhs, rhs) =>
  val ctx1 = genLoad(rhs, ctx, toTypeKind(lhs.symbol.info))
  val Some(l) = ctx.method.lookupLocal(lhs.symbol)
  ctx1.bb.emit(STORE_LOCAL(l), tree.pos)
  ctx1

case _ => /*- ValDef */
  genLoad(tree, ctx, UNIT)
}

```

If left to its own devices, GenICode translates a constructor invocation for a valuetype as for any REFERENCE(cls), i.e. as shown below.

```

private def genLoad(tree: Tree, ctx: Context, expectedType: TypeKind): Context = {
  var generatedType = expectedType
  if (settings.debug.value)
    log("at line: " + (if (tree.pos.isDefined) tree.pos.line else tree.pos))

  val resCtx: Context = tree match {
    . . .
    // 'new' constructor call: Note: since constructors are
    // thought to return an instance of what they construct,
    // we have to 'simulate' it by DUPLICATING the freshly created
    // instance (on JVM, <init> methods return VOID).
    case Apply(fun @ Select(New(tpt), nme.CONSTRUCTOR), args) =>
      val ctor = fun.symbol
      generatedType = toTypeKind(tpt.tpe)
      generatedType match {
        . . .
        case rt @ REFERENCE(cls) =>
          if (settings.debug.value)
            assert(ctor.owner == cls,
              "Symbol " + ctor.owner.fullName + " is different than " + tpt)

          /*- here's where the constructor call is emitted, details below */

          case _ =>
            abort("Cannot instantiate " + tpt + " of kind: " + generatedType)
      }
    . . .
  }
}

```

BTW, the above shows that valuetypes' class symbols are wrapped in REFERENCES just like those for .NET classtypes. Details around this at Sec. 6.2.

By the time control flow gets into "case rt @ REFERENCE(cls) =>" we might be in any of the cases where a classtype or a valuetype is being constructed, either with the default constructor or a parameterful one. In order to share code-emitting code, let's use this building block:

```

/**
 * Adds a local var, the emitted code requires one more slot on the stack as on entry
 */
private def genLoadZeroOfNonEnumValuetype(ctx: Context, kind: TypeKind, pos: Position,
  leaveAddressOnStackInstead: Boolean): Context = {
  val REFERENCE(clssym) = kind
}

```

```

assert(loaders.clrTypes.isNonEnumValuetype(classym))
val local = ctx.makeLocal(pos, classym.tpe, "tmp")
ctx.method.addLocal(local)
ctx.bb.emit(CIL_LOAD_LOCAL_ADDRESS(local), pos)
ctx.bb.emit(CIL_INITOBJ(kind), pos)
val instr = if (leaveAddressOnStackInstead)
    CIL_LOAD_LOCAL_ADDRESS(local)
    else
    LOAD_LOCAL(local)
ctx.bb.emit(instr, pos)
ctx
}

```

The snippet below constitutes the `forJVM` handler for “`case rt @ REFERENCE(cls) =>`”. Its net effect is to leave on the stack a new value:

```

val nw = NEW(rt) /*- no receiver is pushed */
ctx.bb.emit(nw, tree.pos) /*- NEW(REFERENCE(cls)) emitted */
ctx.bb.emit(DUP(generatedType)) /*- DUP emitted */
val ctx1 = genLoadArguments(args, ctor.info.paramTypes, ctx) /*- the args are pushed */

val init = CALL_METHOD(ctor, Static(true)) /*- finally, call to constructor */
nw.init = init
ctx1.bb.emit(init, tree.pos)
ctx1

```

For non-enum valuetypes, we’ll emit code with the same net effect (and consuming no more stack slots, thus we don’t need to update `maxstacksize`). `genLoadZeroOfNonEnumValuetype` does most of the job already, but does not handle parameterful constructors (which are the only custom constructors, a default constructor can’t be defined for valuetypes, CLR dixit). The emitted code contains neither `NEW` nor `DUP` instructions:

```

val ctx2 = if (loaders.clrTypes.isNonEnumValuetype(cls)) {
    /* parameterful constructors are the only possible custom constructors,
       a default constructor can't be defined for valuetypes, CLR dixit */
    val isDefaultConstructor = args.isEmpty
    if (isDefaultConstructor) {
        genLoadZeroOfNonEnumValuetype(ctx, rt, tree.pos, leaveAddressOnStackInstead = false)
        ctx
    } else {
        val ctx1 = genLoadArguments(args, ctor.info.paramTypes, ctx)
        ctx1.bb.emit(CIL_NEWOBJ(ctor), tree.pos)
        ctx1
    }
} else {
    /*- code for classtypes (emits NEW, DUP, loads args, and ctor call ) */
}

```

Finally, what `GenMSIL` does for `CIL_NEWOBJ` is unsurprising:

```

case CIL_NEWOBJ(msym) =>
    assert(msym.isClassConstructor)
    val constructorInfo: ConstructorInfo = getConstructor(msym)
    mcode.Emit(OpCodes.Newobj, constructorInfo)

```


4.6 Recipe for GenMSIL

The idea is to keep GenMSIL as straightforward as possible, i.e., no complex logic about what to emit should end up there (that can be expressed more succinctly in GenICode). For example:

```
case LOAD_LOCAL(local) => loadLocalOrAddress(local, "load_local", loadAddr = false)
case CIL_LOAD_LOCAL_ADDRESS(local) => loadLocalOrAddress(local, "cil_load_local_address", loadAddr = true)
case LOAD_FIELD(field, isStatic) => loadFieldOrAddress(field, isStatic, "load_field", loadAddr = false)
case CIL_LOAD_FIELD_ADDRESS(field, isStatic) => loadFieldOrAddress(field, isStatic, "cil_load_field_address",
```

where `loadLocalOrAddress()` and `loadFieldOrAddress()` exist to avoid code duplication, but do little more than emitting the actual opcodes.

Other cases (`CIL_NEWOBJ`) were discussed in-line alongside their GenICode discussion in the previous section.

5 System.Decimal has operator overloads and conversion operators

Certain CLR method signatures define *operator overloads*. These methods must be public, static, and flagged as `SpecialName` so that they do not collide with the users name space. The list of unary (§I.10.3.1) and binary operators (§I.10.3.2) amenable to this technique is cast in stone by the CIL spec. The following conveys the idea how metadata-level names are mapped to operator symbols:

```
// from TypeParser.scala
private def getName(method: MethodBase): Name = {

  def operatorOverload(name : String, paramsArity : Int) : Option[Name] = paramsArity match {
    case 1 => name match {
      // PartitionI.10.3.1
      case "op_Decrement" => Some(encode("--"))
      case "op_UnaryNegation" => Some(nme.UNARY_-)
      . . .
      case _ => None
    }
    . . .
  }
}
```

Quoting from §I.10.3.3 Conversion operators:

Conversion operators are unary operations that allow conversion from one type to another. The operator method shall be defined as a static method on either the operand or return type. There are two types of conversions:

- *An implicit (widening) coercion shall not lose any magnitude or precision. These should be provided using a method named `op_Implicit`.*
- *An explicit (narrowing) coercion can lose magnitude or precision. These should be provided using a method named `op_Explicit`.*

```

// from TypeParser.scala
/*- remember, there's typ.getMethods and type.GetMethods */
for (method <- typ.getMethods)
  if (!method.HasPtrOrByRefParamOrRetType &&
      method.IsPublic && method.IsStatic && method.IsSpecialName &&
      method.Name == "op_Implicit") {
    // create a view: typ => method's return type
    val viewRetType: Type = sig2type(method.ReturnType)
    val viewParamTypes: List[Type] = method.GetParameters().map(_.ParameterType).map(getCLSType).toList;
    /* The spec says "The operator method shall be defined as a static method on either the operand or return type
     * We don't consider the declaring type for the purposes of definitions.functionType,
     * instead we regard op_Implicit's argument type and return type as defining the view's signature.
     */
    if (viewRetType != null && !viewParamTypes.contains(null)) {
      /* The check above applies e.g. to System.Decimal that has a conversion from UInt16,
       * a non-CLS type, whose CLS-mapping returns null */
      val funType: Type = definitions.functionType(viewParamTypes, viewRetType);
      val flags = Flags.JAVA | Flags.STATIC | Flags.IMPLICIT; // todo: static? shouldn't be final instead?
      val viewMethodType = (msym: Symbol) => JavaMethodType(msym.newSyntheticValueParams(viewParamTypes), funType)
      val vmsym = createMethod(nme.view_, flags, viewMethodType, method, true);
      methodsSet -= method;
    }
  }
}

```

```

Variables
this = {scala.tools.nsc.symtab.SymbolLoaders$MSILTypeLoader$typeParser$@2002}
global = {scala.tools.nsc.Global@2020}
scala$tools$nsc$symtab$clr$typeParser$$dazz = {scala.tools.nsc.symtab.Symbols$ClassSymbol@2004}"class Decimal"
scala$tools$nsc$symtab$clr$typeParser$$instanceDefs = {scala.tools.nsc.symtab.Scopes$Scope@2021}"Scope({n <rn
staticModule = {scala.tools.nsc.symtab.Symbols$ModuleSymbol@2022}"object Decimal"
scala$tools$nsc$symtab$clr$typeParser$$staticDefs = {scala.tools.nsc.symtab.Scopes$Scope@2023}"Scope({n final

Value=Scope{
final <java> <static> val Zero: System.this.Decimal;
final <java> <static> val One: System.this.Decimal;
final <java> <static> val MinusOne: System.this.Decimal;
final <java> <static> val MaxValue: System.this.Decimal;
final <java> <static> val MinValue: System.this.Decimal;
implicit <method> <java> <static> def view(<param> <synthetic> x$1: System.this.Byte): scala.this.Function1[System.this.Byte,System.
implicit <method> <java> <static> def view(<param> <synthetic> x$1: scala.this.Byte): scala.this.Function1[scala.this.Byte,System.this.
implicit <method> <java> <static> def view(<param> <synthetic> x$1: scala.this.Short): scala.this.Function1[scala.this.Short,System.this.
implicit <method> <java> <static> def view(<param> <synthetic> x$1: scala.this.Char): scala.this.Function1[scala.this.Char,System.this.
implicit <method> <java> <static> def view(<param> <synthetic> x$1: scala.this.Int): scala.this.Function1[scala.this.Int,System.this.Dec
implicit <method> <java> <static> def view(<param> <synthetic> x$1: scala.this.Long): scala.this.Function1[scala.this.Long,System.this.
<method> <java> <static> def ToByte(<param> <synthetic> x$1: System.this.Decimal): scala.this.Byte;
<method> <java> <static> def Parse(<param> <synthetic> x$1: System.this.String): System.this.Decimal;
<method> <java> <static> def op_Explicit(<param> <synthetic> x$1: System.this.Decimal): scala.this.Int;
<method> <java> <static> def Floor(<param> <synthetic> x$1: System.this.Decimal): System.this.Decimal;
<method> <java> <static> def ToInt32(<param> <synthetic> x$1: System.this.Decimal): scala.this.Int;
<method> <java> <static> def ToInt16(<param> <synthetic> x$1: System.this.Decimal): scala.this.Short;
<method> <java> <static> def +(<param> <synthetic> x$1: System.this.Decimal, <param> <synthetic> x$2: System.this.Decimal): Syst
<method> <java> <static> def FromOACurrency(<param> <synthetic> x$1: scala.this.Long): System.this.Decimal;
<method> <java> <static> def Parse(<param> <synthetic> x$1: System.this.String, <param> <synthetic> x$2: Globalization.this.Numbe
<method> <java> <static> def <=<(<param> <synthetic> x$1: System.this.Decimal, <param> <synthetic> x$2: System.this.Decimal): sca
<method> <java> <static> def Round(<param> <synthetic> x$1: System.this.Decimal): System.this.Decimal;
<method> <java> <static> def op_Implicit(<param> <synthetic> x$1: System.this.UInt32): System.this.Decimal;
<method> <java> <static> def ++(<param> <synthetic> x$1: System.this.Decimal): System.this.Decimal;
<method> <java> <static> def %(<param> <synthetic> x$1: System.this.Decimal, <param> <synthetic> x$2: System.this.Decimal): Syst

```

TODO: interface name should be stripped from method name (when intered into Scope) however in MethodInfo should remain "non-stripped".

TODO no field in MethodInfo gives the explicitly overridden MethodInfo (if any).

6 Sidenotes

6.1 How are value types implemented in the 32-bit CLR?

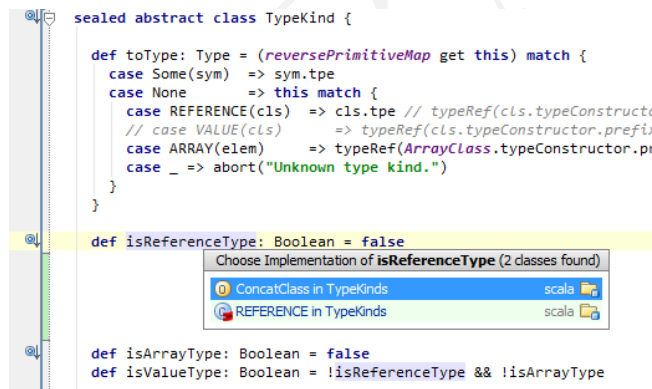
<http://blogs.msdn.com/b/clrcodegeneration/archive/2007/11/02/how-are-value-types-implemented.aspx>

6.2 GenICode: valuetype class symbols are wrapped in REFERENCES just like those for classtypes

This causes no problem as long as we don't step into trouble. Invoking `zeroOf()` for a `REFERENCE(valuetype)` would lead to trouble:

```
def zeroOf(k: TypeKind): Tree = k match {
  case UNIT      => Literal(())
  case BOOL      => Literal(false)
  case BYTE      => Literal(0: Byte)
  case SHORT     => Literal(0: Short)
  case CHAR      => Literal(0: Char)
  case INT       => Literal(0: Int)
  case LONG      => Literal(0: Long)
  case FLOAT     => Literal(0.0f)
  case DOUBLE    => Literal(0.0d)
  case REFERENCE(cls) => Literal(null: Any)
  case ARRAY(elem)  => Literal(null: Any)
  case BOXED(_)    => Literal(null: Any)
  case ConcatClass => abort("no zero of ConcatClass")
}
```

BTW, `TypeKind.isValueType` has nothing to do with .NET valuetypes:



```
sealed abstract class TypeKind {
  def toType: Type = (reversePrimitiveMap get this) match {
    case Some(sym) => sym.type
    case None      => this match {
      case REFERENCE(cls) => cls.type // typeRef(cls.typeConstructor)
      // case VALUE(cls)   => typeRef(cls.typeConstructor.prefix)
      case ARRAY(elem)   => typeRef(ArrayClass.typeConstructor.prefix)
      case _              => abort("Unknown type kind.")
    }
  }
}

def isReferenceType: Boolean = false
def isArrayType: Boolean = false
def isValueType: Boolean = !isReferenceType && !isArrayType
```

The screenshot shows a code editor with the above Scala code. A tooltip is visible over the `isReferenceType` property, showing two possible implementations: `ConcatClass in TypeKinds` and `REFERENCE in TypeKinds`, both from the `scala` package.

6.3 GenMSIL: unboxing of Booleans without System.Convert

In `GenMSIL.emitUnbox` most conversions invoke utilities in `System.Convert`, except:

```
def emitUnbox(code: ILGenerator, boxType: TypeKind) = (boxType: @unchecked) match {
  case UNIT => code.Emit(OpCodes.Pop)
  case BOOL => code.Emit(OpCodes.Unbox, MBOOL); code.Emit(OpCodes.Ldind_I1)
  . . .
}
```

For the record, details about the idiom “`unbox typeTokOfValueType; ldind.i;`”:

Unboxing: `unbox typeTokOfValueType`. Stack transition:

```
..., obj -> ..., valueTypePtr
```

The `unbox` instruction converts `obj` (of type `0`), the boxed representation of a value type, to `valueTypePtr` (a controlled-mutability managed pointer, type `&`), its unboxed form. The type of the value type contained within `obj` must be assignment compatible with `valueType`. *Note: This effects the behavior with enum types, see Partition II.14.3.* Unlike `box`, which is required to make a copy of a value type for use in the object, `unbox` simply computes the address of the value type that is already present inside of the boxed object.

Indirect load of a byte: `ldind.i1`. Stack transition:

```
..., addr -> ..., value
```

6.4 Controlled-mutability managed pointers

Quoting from §III.1.8.1.2.2 *Controlled-mutability managed pointers*:

The `readonly` prefix and `unbox` instructions can produce what is called a controlled-mutability managed pointer. Unlike ordinary managed pointer types, a controlled-mutability managed pointer is incompatible with ordinary managed pointers; e.g., it cannot be passed as a `byref` argument to a method. At control flow points, a controlled-mutability managed pointer can be merged with a managed pointer of the same type to yield a controlled-mutability managed pointer.

Controlled-mutability managed pointers can only be used in the following ways:

- *As the object parameter for an `ldfld`, `ldflda`, `stfld`, `call`, `callvirt`, or `constrained.callvirt` instruction.*
- *As the pointer parameter to a `ldind.*` or `ldobj` instruction.*
- *As the source parameter to a `cpobj` instruction.*

All other operations (including `stobj`, `stind.*`, `initobj`, and `mkrefany`) are invalid.

The pointer is called a controlled-mutability managed pointer because the defining type decides whether the value can be mutated. For value classes that expose no public fields or methods that update the value in place, the pointer is read-only (hence the name of the prefix). In particular, the classes representing primitive types (such as `System.Int32`) do not expose mutators and thus are read-only.

6.5 Emitting (un)boxing in GenMSIL

```
def emitBox(code: ILGenerator, boxType: TypeKind) = (boxType: @unchecked) match {  
  // doesn't make sense, unit as parameter..  
  case UNIT => code.Emit(OpCodes.Ldsfld, boxedUnit)  
  case BOOL | BYTE | SHORT | CHAR | INT | LONG | FLOAT | DOUBLE =>
```

```

    code.Emit(OpCodes.Box, msilType(boxType))
  case REFERENCE(cls) if (definitions.unboxMethod.contains(cls)) =>
    /*- unboxMethod.keys and boxMethod.keys contain the same, right? */
    code.Emit(OpCodes.Box, (msilType(boxType)))
    /*- msilType(boxType) provides the typeTok that the box instruction expects,
       which in turn should be compatible with the value on the stack.
       typeTok can indicate a value type (in particular a nullable type)
       or a reference type (that's why 'box typeTok' may push null too)
       or a type param. However, 'box typeTok' can't consume a managed pointer from the stack. */
  case REFERENCE(_) | ARRAY(_) => ()
}

def emitUnbox(code: ILGenerator, boxType: TypeKind) = (boxType: @unchecked) match {
  case UNIT => code.Emit(OpCodes.Pop)
  case BOOL => code.Emit(OpCodes.Unbox, MBOOL); code.Emit(OpCodes.Ldind_I1)
  case BYTE => code.Emit(OpCodes.Call, toByte)
  case SHORT => code.Emit(OpCodes.Call, toShort)
  case CHAR => code.Emit(OpCodes.Call, toChar)
  case INT => code.Emit(OpCodes.Call, toInt)
  case LONG => code.Emit(OpCodes.Call, toLong)
  case FLOAT => code.Emit(OpCodes.Call, toFloat)
  case DOUBLE => code.Emit(OpCodes.Call, toDouble)
  case REFERENCE(cls) if (definitions.unboxMethod.contains(cls)) =>
    code.Emit(OpCodes.Unbox, msilType(boxType)) /*- leaves a valueTypePtr (managed pointer) on the stack */
    code.Emit(OpCodes.Ldobj, msilType(boxType)) /*- consumes the address and pushes the raw value type instance.
                                                    ldobj also accepts as typeToken a type param. */
  case REFERENCE(_) | ARRAY(_) => ()
}

```

6.6 “Unavoidable boxing” due to event declarations on structs

“Binding delegate to a struct causes the struct to be boxed. So, expression *xo.Xhandler* evaluates to different delegate instances.”. More at <http://social.msdn.microsoft.com/Forums/en-US/csharp/language/thread/11335595-4ca9-4598-ad83-d031>

7 TODO

7.1 Passing a literal field to a byref argument should trigger an error message

Passing a literal field to a byref argument should trigger an error message (from typer better than from GenICode)

```
ColorDemo.rcvColor(Color.Red)
```

As of now, it's detected only in GenMSIL, and with an assert at that:

```

def loadFieldOrAddress(field: Symbol, isStatic: Boolean, msg: String, loadAddr : Boolean) {
  var fieldInfo = fields.get(field) match {
    case Some(fInfo) => fInfo
    case None =>
      val fInfo = getType(field.owner).GetField(msilName(field))
      fields(field) = fInfo
      fInfo
  }
}

```

```

}
if (!fieldInfo.IsLiteral) {
    . . .
} else {
    assert(!loadAddr, "can't take AddressOf a literal field (not even with readonly prefix) because no memory
    . . .

```

7.2 More TODO's

- `System.SByte` is type-parsed with a type symbol different from that in `definitions.ByteClass`. Type members are entered afterwards not in `definitions.ByteClass`. Is this good or bad?
- currently `System.UInt32` is type-parsed, but not clear whether we want to handle it.
- */* TODO IsByRef should be tracked for the param definition, not for its type */*
- add code to handle `UNARY_~` and `sizeof` for valuetypes and enums.
- check in `GenMSIL` whether, for `sym` an interface, the following overlooks one super-interface:

```
val interfaces: Array[MsilType] = parents.tail.map(p => msilTypeFromSym(p.typeSymbol)).toArray
```

References

- [1] Microsoft Corporation. C# version 3.0 language specification, 2007. <http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx>.

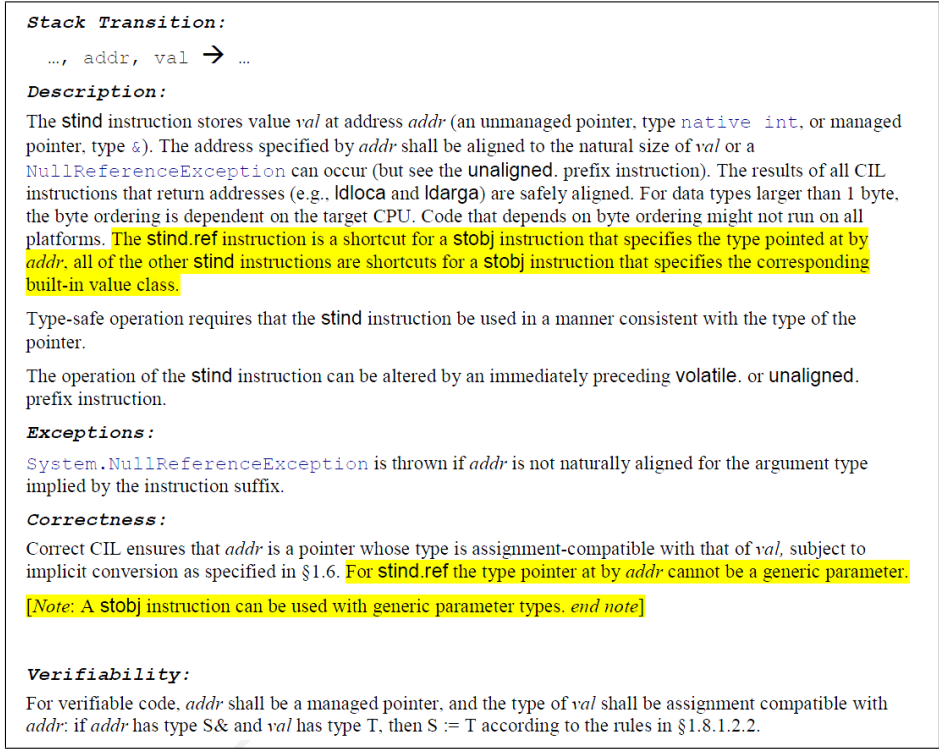


Figure 2: `stind` story (Sec. 3.6)

Format	Assembly Format	Description
81 <T>	Stobj <i>typeTok</i>	Store a value of type <i>typeTok</i> at an address.

Stack Transition:
..., *dest*, *src* → *...*

Description:
 If *typeTok* is a value type, the **stobj** instruction copies the value *src* to the address *dest*. If *typeTok* is not a generic parameter and either a reference type or a built-in value class, then the **stind** instruction provides a shorthand for the **stobj** instruction.

The operation of the **stobj** instruction can be altered by an immediately preceding **volatile**. or **unaligned**. prefix instruction.

Exceptions:
`System.NullReferenceException` can be thrown if an invalid address is detected.
`System.TypeLoadException` is thrown if *typeTok* cannot be found. This is typically detected when CIL is converted to native code rather than at runtime.

Correctness:
typeTok shall be a valid `typedef`, `typeref`, or `typespec` metadata token.

[Note: Unlike the **stind** instruction a **StObj** instruction can be used with a generic parameter type. *end note*]

Verifiability:
 Let the static type of the value on top of the stack be some type *srcType*. The value shall be initialized (when *srcType* is a reference type). The static type of the destination address *dest* on the preceding stack slot shall be a managed pointer (of type `destType &`) to some type *destType*. Finally, *srcType* shall be assignment-compatible with *typeTok*, and *typeTok* shall be a subtype of *destType*. When *typeTok* is a non-reference type, the definition of subtyping implies that *srcType* shall be assignment-compatible with *destType* (which, itself, shall be equal to *typeTok*).

Figure 3: **stobj** story (Sec. 3.6)

Listing 1: Inlining of literal fields (Sec. 1.3)

```

case LOAD_FIELD(field, isStatic) =>
  if (settings.debug.value)
    log("LOAD_FIELD with owner: " + field.owner +
        " flags: " + Flags.flagsToString(field.owner.flags))
  var fieldInfo = fields.get(field) match {
    case Some(fInfo) => fInfo
    case None =>
      val fInfo = getType(field.owner).GetField(msilName(field))
      fields(field) = fInfo
      fInfo
  }
  if (!fieldInfo.IsLiteral) {
    mcode.Emit(if (isStatic) OpCodes.Ldsfld else OpCodes.Ldfld, fieldInfo)
  } else {
    /* emit as for a CONSTANT ICode stmt, with the twist that the constant value is available
    * as a java.lang.Object and its .NET type allows constant initialization in CLR, i.e. that type
    * is one of I1, I2, I4, I8, R4, R8, CHAR, BOOLEAN, STRING, or CLASS (in this last case,
    * only accepting nullref as value). See Table 9-1 in Lidin's book on ILAsm. */
    val value = fieldInfo.getValue()
    if (value == null) {
      mcode.Emit(OpCodes.Ldnull)
    } else {
      val typ = if (fieldInfo.FieldType.IsEnum) fieldInfo.FieldType.getUnderlyingType
                else fieldInfo.FieldType
      if (typ == clrTypes.STRING) {
        mcode.Emit(OpCodes.Ldstr, value.asInstanceOf[String])
      } else if (typ == clrTypes.BOOLEAN) {
        mcode.Emit(if (value.asInstanceOf[Boolean]) OpCodes.Ldc_I4_1
                     else OpCodes.Ldc_I4_0)
      } else if (typ == clrTypes.BYTE || typ == clrTypes.UBYTE) {
        loadI4(value.asInstanceOf[Byte], mcode)
      } else if (typ == clrTypes.SHORT || typ == clrTypes.USHORT) {
        loadI4(value.asInstanceOf[Int], mcode)
      } else if (typ == clrTypes.CHAR) {
        loadI4(value.asInstanceOf[Char], mcode)
      } else if (typ == clrTypes.INT || typ == clrTypes.UINT) {
        loadI4(value.asInstanceOf[Int], mcode)
      } else if (typ == clrTypes.LONG || typ == clrTypes.ULONG) {
        mcode.Emit(OpCodes.Ldc_I8, value.asInstanceOf[Long])
      } else if (typ == clrTypes.FLOAT) {
        mcode.Emit(OpCodes.Ldc_R4, value.asInstanceOf[Float])
      } else if (typ == clrTypes.DOUBLE) {
        mcode.Emit(OpCodes.Ldc_R4, value.asInstanceOf[Double])
      } else {
        abort("Unknown type for static literal field: " + fieldInfo)
      }
    }
  }
}
}

```

Listing 2: How GenICode replaces some callsites with a BOX instruction

```

/**
 * Generate code for trees that produce values on the stack
 *
 * @param tree The tree to be translated
 * @param ctx The current context
 * @param expectedType The type of the value to be generated on top of the
 *                    stack.
 * @return The new context. The only thing that may change is the current
 *         basic block (as the labels map is mutable).
 */
private def genLoad(tree: Tree, ctx: Context, expectedType: TypeKind): Context = {
  var generatedType = expectedType
  if (settings.debug.value)
    log("at line: " + (if (tree.pos.isDefined) tree.pos.line else tree.pos))

  val resCtx: Context = tree match {

    . . .

    case Apply(fun @ _, List(expr)) if (definitions.isBox(fun.symbol)) =>
      if (settings.debug.value)
        log("BOX : " + fun.symbol.fullName);
      val ctx1 = genLoad(expr, ctx, toTypeKind(expr.tpe))
      val nativeKind = toTypeKind(expr.tpe)
      if (settings.Xdce.value) {
        // we store this boxed value to a local, even if not really needed.
        // boxing optimization might use it, and dead code elimination will
        // take care of unnecessary stores
        var loc1 = ctx.makeLocal(tree.pos, expr.tpe, "boxed")
        ctx1.bb.emit(STORE_LOCAL(loc1))
        ctx1.bb.emit(LOAD_LOCAL(loc1))
      }
      ctx1.bb.emit(BOX(nativeKind), expr.pos)
      generatedType = toTypeKind(fun.symbol.tpe.resultType)
      ctx1

    . . .

    case EmptyTree =>
      if (expectedType != UNIT)
        ctx.bb.emit(getZeroOf(expectedType))
      ctx

    case _ =>
      abort("Unexpected tree in genLoad: " + tree + " at: " + tree.pos)
  }

  // emit conversion
  if (generatedType != expectedType)
    adapt(generatedType, expectedType, resCtx, tree.pos)

  resCtx
}

```
